Enhancing NIC Performance for MPI using Processing-in-Memory

Arun Rodrigues, Richard Murphy, Ron Brightwell, and Keith D. Underwood
Sandia National Laboratories*
P.O. Box 5800, MS-1110
Albuquerque, NM 87185-1110
{afrodri, rcmurph, rbbright, kdunder}@sandia.gov

Abstract

Processing-in-Memory (PIM) technology encompasses a range of research leveraging a tight coupling of memory and processing. The most unique features of the technology are extremely wide paths to memory, extremely low memory latency, and wide functional units. Many PIM researchers are also exploring extremely fine-grained multi-threading capabilities. This paper explores a mechanism for leveraging these features of PIM technology to enhance commodity architectures in a seemingly mundane way: accelerating MPI. Modern network interfaces leverage simple processors to offload portions of the MPI semantics. particularly the management of posted receive and unexpected message queues. Without adding cost or increasing clock frequency, using PIMs in the network interface can enhance performance. The results are a significant decrease in latency and increase in small message bandwidth, particularly when long queues are present.

1. Introduction

Processing-in-Memory (PIM) [5, 13] (or Intelligent RAM [19]) is a novel technology that is receiving widespread attention in high-performance computing. PIMs merge a processor with memory to avoid the impending memory wall [28]. Researchers envision a future with a sea of memory with processors scattered throughout [13, 5]. Those processors would be simple with wide paths to memory and wide functional units [4]. They would be inherently multithreaded [23] with extremely lightweight thread context and thread invocation mechanisms. This paper, however, is not about revolutionizing supercomputer architecture with PIMs. It is not going to discuss how PIMs may completely change the memory system. Instead, this paper focuses on how PIMs could transform supercomputers in the least expected of places: the network interface.

Modern network interfaces offload much of the MPI processing [1, 20]. This is a natural migration in light of research indicating that network overhead (the time the application processor spends handling messages traffic [10]) has the single largest impact on application performance [15]. Unfortunately, the work that is offloaded includes traversing the posted receive queue, which can grow quite long [3]. As the queue grows, the time to traverse it also grows due to the inherently low clock speed and simplicity of the processor on the NIC [25]. Thus, another important measure of network performance, the gap (time between when new messages can be initiated [10]) begins to grow. Attempts to minimize this traversal time through hashing have been considered and abandoned by vendors due to the increase in list insertion time and the existence of wildcarded calls to MPI in several codes.

An alternative solution is to build a better NIC processor. Such a processor needs to decrease the time to search a linked list to reduce the latency of messages. It needs to overlap multiple searches of the list to increase throughput. It needs to... look a lot like a PIM. A PIM has low latency access to memory, so it traverses linked lists well. Memory access is wide and can be operated on with a wide ALU, so multiple list entries can be searched rapidly with a handful of instructions. The PIM is multi-threaded with extremely fine-grained locking support, so simultaneous queue traversals can be used to leverage the increase in bandwidth. Best of all, PIMs are simple, so the area and the NRE devoted to them will be minimal. In short, PIMs are a natural fit to the needs of the embedded processor on the NIC.

This paper explores the details of how a PIM can accelerate NIC-based MPI processing. A model of a modern NIC was created as a baseline for comparison. In the standard NIC, a PowerPC 440 embedded processor was used. A PIM based on similar technology was also used. An MPI

* Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

implementation was modified to leverage the wide memory accesses, the wide ALU operations, and the multithreading capabilities of the PIM. The result was a dramatic decrease in message latency in the presence of long queues and an increase in the achievable throughput.

In Section 2, related work is described. Following that the hardware (Section 3) and MPI implementation (Section 4) are described. The evaluation methodology is described in Section 5 and the results are presented in Section 6. The paper wraps up with conclusions and future work in Section 7.

2. Related Work

This paper explores PIM-based acceleration of match processing for the MPI posted receive queue. This is a natural progression of previous work on utilizing PIM hardware to support MPI and exploring those areas of MPI that may benefit from hardware assisted processing. We have previously addressed issues in supporting MPI functionality for a PIM-based system architecture [22]. We have also proposed augmenting a traditional network interface with custom hardware for accelerating MPI queue processing [26]. To our knowledge, there is no previous work on strategies for optimizing MPI queue processing in published research.

In contrast, a significant amount of previous work is aimed at utilizing programmable network interface hardware, such as Quadrics [20] and Myrinet [2], for optimizing MPI performance. These optimizations include offloading MPI protocol processing [24, 14], using hardware capabilities for efficient data movement (especially collective operations [7, 17]), and using scatter/gather functionality for handling non-contiguous data transfers [27].

As a processing technology, PIM has been examined for use in supercomputing for almost a decade[13, 23, 18, 5, 4]. It has also been examined in detail in the context of embedded systems[12]. However, to date, we know of no analysis of PIMs used to enhance NIC performance. Given the high bandwidth access to memory made possible by a PIM, combined with the throughput-oriented (latency tolerant) architecture proposed in the context of supercomputing, accelerating the performance of MPI processing seems like a natural match.

3. Hardware Overview

The baseline NIC (shown in Figure 1) is representative of numerous modern network interfaces, including the Quadrics [20] network, the Myrinet network [2], and the Red Storm system developed by Cray and Sandia [1, 9]. These NICS interface to the network fabric on one side and a high-performance interface to the microprocessor (e.g. HyperTransport or PCI-Express) on the other. What delivers the high-performance is the logic between the two interfaces: two DMA engines (one in each direction) driven by local processing on the network interface. The processing on the network interface is a microprocessor (or two) with either internal RAM (as shown in Figure 1(a) and found in Red Storm [1, 9]) or external RAM (as is typical in commodity networks like Myrinet [2] and Quadrics [20]).

3.1. PIM-Based NIC Enhancement

Commodity processors are limited by their architecture. They can only process one input at a time (they are single threaded), and they have narrow functional units requiring many instructions to process a single list entry. In contrast, PIMs can use fine-grain locking and multithreading support to perform concurrent list walks for multiple messages simultaneously. They also have wide functional units that allow them to compare multiple list entries simultaneously. PIMs use reduced core complexity (yielding slightly lower instruction per cycle (IPC) rate for some workloads) to dramatically reduce core size. Thus, PIMs have the potential to bring significant gains in message latency and message throughput when the posted receive queue is long.

The proposed modification is shown in Figure 1(b). The embedded SRAM becomes part of the PIMs and is accessed in extremely wide words. The PIMs interface to the rest of the NIC in exactly the same way as a conventional processor would and have the exact same amount of memory backing them. A single PIM, however, is much smaller. Thus, it is possible to include two PIMs. Each of these PIMs has equal wide access to both memories (contention is now possible) as indicated by the wide connection between the SRAMs. Virtually all PIM architectures consider multiple cooperating PIMs in a single memory, so two PIMs is a reasonable design point.

3.2. Relative Area Requirements

Silicon is still a precious resource and is the standard for comparing two NIC designs. It is only fair to compare two designs if they are of comparable area. Thus, it is critical to understand the area trade-off when comparing a PIM-based NIC to a conventional NIC. As the first significant PIM system to incorporate the essential features examined, PIM Lite [4] provides the hard data needed for comparison. Brockman [6] compares the relative area of conventional processor cores, PIM Lite, and conventional memories in a process-independent fashion. The 128-bit PIM Lite implementation has an equivalent area to 10.3 KB of SRAM. Since this paper uses a 256-bit data path, the area is doubled to 20.6 KB of SRAM. Assuming that additional features might be desired in the PIM core, we double this esti-

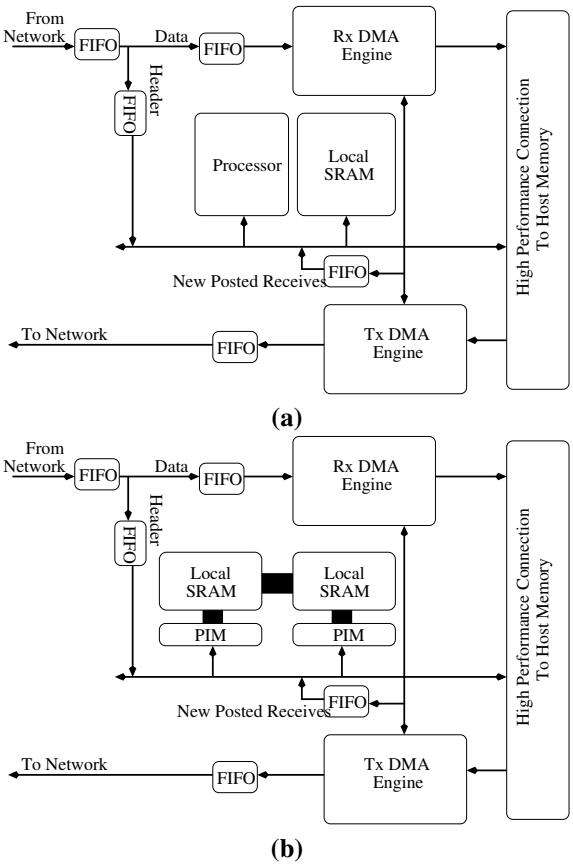

Figure 1. A high-performance network interface enhanced with (a) a processor, and (b) multiple PIMs

mate and add a 4 KB instruction cache to yield a 45.2 KB SRAM equivalence. The design evaluated here uses two PIM cores (90.4 KB of SRAM equivalence) and 384K of SRAM. By comparison, the PowerPC 440 integer core is equivalent to 33.3 KB of SRAM [6]. It includes 32 KB of instruction cache and 32 KB of data cache for a total of 97.3 KB of SRAM equivalence. Thus, the PIM solution is only 93% of the area that is required by the PowerPC solution.

4. MPI Implementation

To explore MPI acceleration, a prototype MPI implementation that was split between the host processor and the NIC processor was created. A split implementation of MPI, rather than an implementation over a more general purpose API, was created to maximize performance. The prototype MPI implements a subset of MPI-1.2 [16]. With the exception of `MPI_Barrier()`, only basic point-to-point communication and basic support functions were implemented(Figure 2). Only support for basic MPI Datatypes is included and `MPI_COMM_WORLD` is the only group. The MPI implementation is roughly 1600 lines of C++ and is compiled with GNU g++ 3.3.

```
MPI_Comm_rank()        MPI_Isend()
MPI_Comm_size()        MPI_Recv() †
MPI_Finalize()         MPI_Send() †
MPI_Init() †           MPI_Wait()
MPI_Irecv()            MPI_WaitAll() †
MPI_Barrier() †
```

Figure 2. Subset of MPI implemented †*functions built from other MPI functions*

4.1. Host Software

The host portion of the MPI implementation is minimal. The three basic communication functions (`MPI_Irecv()`, `MPI_Isend()`, `MPI_Wait()`) offload virtually all work onto the network interface. The functions built on top of those as well as the non-communication functions are implemented on the host. To perform an `MPI_Irecv()`, the host checks for available space and posts an item to a queue in the network interface. The network interface manages the posted receive queue and unexpected message queue. The implementation of `MPI_Isend()` is similar. It only needs to place information about the message to be sent in a queue on the network interface. To implement an `MPI_Wait`, the host spins on a memory location in host memory.

4.2. NIC-Based MPI Implementation

Most of the work is done by the processor on the NIC. It manages all NIC resources, drives the DMA engines, and handles most of the MPI semantics (including progress and matching). To accomplish this, the NIC maintains five linked lists of MPI state. The following lists contain requests and the state required to advance them.

- `postedRecvQ`: Posted receive buffers for incoming messages to match against
- `activeRecvQ`: Active receives requiring processing (i.e. rendezvous requests which need a reply, requests waiting for a DMA engine, etc.)
- `unexpectedQ`: List of unexpected messages. Compared to new posted receives.
- `unexpectedActiveQ`: Active unexpected messages which must be advanced (i.e. unexpected messages requiring DMA transfer).

- `sendQ`: Queue send requests for processing.

The core of the software on the network interface processor is a loop that continually checks for work. If a new message is waiting to be processed, the header is read by the NIC processor. The posted receive queue is traversed and the header is compared to each posted receive. Upon finding a match, the processor moves the receive request to the active list and either a DMA is setup (a short message) or a rendezvous reply is sent (a long message). If no match is found, the message is placed on the `unexpectedQ`, to be compared to future receives as they are posted.

New send requests cause either a DMA to start (for short messages) or a rendezvous request to be sent (for long messages). The send request is then added to the list of active requests. New receive requests are first compared to the existing unexpected message queue (`unexpectedQ`) for a match. A match causes the message to be copied appropriately or, for long messages, for the data to be requested from the remote processor. Failure to match the receive with the `unexpectedQ` will cause the receive to be placed on the `postedRecvQ`.

The processor also checks the active queues (`activeRecvQ` and `unexpectedActiveQ`) for messages that need to be progressed. Progressing messages can include such things as providing additional information to the DMA engine and responding to rendezvous replies. This enables the MPI implementation to meet the strictest interpretation of the independent progress rule while using a rendezvous protocol and without involving a host processor thread. As items on the active queues complete, the host processor is notified by writing a location in host memory.

4.3. PIM Feature Exploitation

PIMs have several features that enable higher performing NICs. The most prominent of these is the extremely wide path between the memory and the processor core (256 bits rather than 32 or 64 bits). This path enables the processor to load large pieces of data to be matched with a single instruction. Moreover, PIMs do not have traditional caches, which typically force several sequential transfers from memory to cache (regardless of access size) and cause unneeded data to be loaded along with the requested data.

The second useful feature of PIMs is a wide ALU. Since PIMs have wide paths between processor and memory, they provide an ALU that matches that width. Thus, the entire match (often multiple bytes) can be performed in a smaller number of instructions. Since MPI matches use relatively little data, this is a seemingly small advantage; however, a change in the organization of the list data structures enables a much more powerful usage of the wide ALU. Figure 3 illustrates this concept.

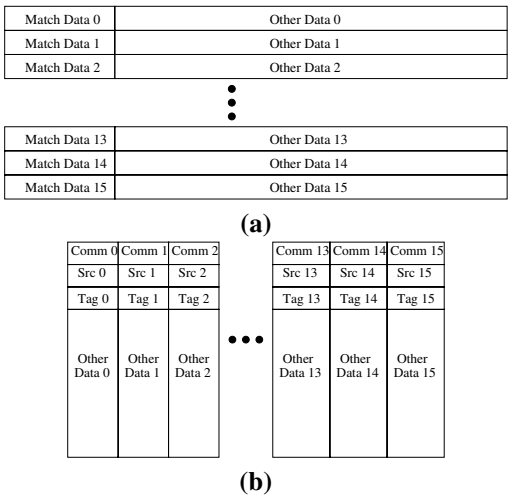

Figure 3. (a) A conventional data layout; (b) a PIM data layout

In Figure 3(a), the typical data layout of an MPI posted receive queue is shown. Memory addresses increase from right to left and from top to bottom. Although the queue entries are often sequential (due to the way list items are allocated), that has no impact on this example. If the MPI match data is 48 bits (16 bits for each of the three fields — tag, source, and communicator — is sufficient to meet the specification), then six bytes of each cache line are useful. Since cache lines are 32 (or more) bytes and processors fetch a full cache line for each cache "miss", much of the data retrieved from memory is wasted. Sixteen memory accesses would be required to examine sixteen queues. With the data organization shown in Figure 3(a), the same would occur for the wide words accessed by the PIM.

The best data layout for a PIM, however, is shown in Figure 3(b), where the data structures are interleaved. A single list entry has sixteen list items. All 256 bits of the PIM access to memory is used and only three memory accesses are needed to examine sixteen list items. This utilizes the wide ALU more effectively because all of the data bits are needed for matching. The obvious question is: could a processor leverage the same data layout? The realistic answer is no. In real usage, the list becomes fragmented and any given list entry (with sixteen list items) would contain only a few valid items. The PIM does not suffer from this limitation because of the combination of multithreading and extremely fine-grained synchronization. These two features enable a small fraction of the processing power to be continuously devoted to "list clean-up". A "list clean-up" thread traverses the list using locking at the level of the list entry and compacts out the holes that have accumulated. Thus, at any given time, only a small amount of fragmentation exists

anywhere in the list. Symmetric multi-threading (SMT) microprocessors, by contrast, do not have the fine-grained synchronization primitives needed for a "list clean-up" thread.

Having multiple threads traversing the list using locking at the level of the list entry yields advantages in message throughput as well. For each message, a commodity processor must access the header over a bus with a 20 ns latency. Then, it must traverse a linked list. The processor is only free to move onto the next header when the list traversal is done. In a multithreaded environment, however, one thread can be responsible for reading headers and spawning additional threads to traverse the lists. This effectively hides the latency of accessing the queue to retrieve a header.

5. Methodology

This research is focused on the solution to an interesting problem: how can the latency and throughput impacts of long posted receive queues be reduced? Exploring that question required that a benchmark be developed to quantify the problem. After developing a quantitative understanding of the issues, it was necessary to create an environment where solutions to the issue could be explored. This environment included a model of the baseline NIC as well as a model of the enhanced NIC.

5.1. Benchmarks

The primary motivation for this design was to reduce the latency of and increase the throughput for messages when long posted receive queues were present. The magnitude of the problem was revealed in an earlier study [25] using a newly designed benchmark. This benchmark was extended to study the impacts of the using a PIM in the NIC.

The benchmark designed to measure the impact of changes in the pre-posted receive queue length provides three degrees of freedom to enable the user to measure the impacts of both the receive queue length (affects caching) and of actual queue traversal (affects processing time). This benchmark was also extended to measure the impact of long posted receive queues on message throughput. Using a traditional processor on a NIC, only one incoming message can be handled at a time. Thus, as the length of a queue grows, the number of messages that can be handled per second decreases. This shifts the standard bandwidth curve to the right — at a given message size, the bandwidth is decreased.

The message throughput can actually be decreased independently of decreasing latency. A $1\mu s$ latency that is concentrated in one atomic block implies a limit on message throughput of one million messages per second. If that $1\mu s$ latency can be broken into five independent pipeline stages, then five million messages per second can be achieved. Alternatively, if five parallel processing units can be provided, the same effect can be achieved. To measure this effect, it is necessary to have more than one message in flight. Thus, the benchmark from [25] was modified to have multiple messages (25) in flight. The modifications to the benchmark are shown in Figure 4. This scenario is reasonable for applications that have long posted receive queues, especially if they would normally communicate with multiple neighbors.

```
prepost_traversed_receives();
post_25_latency_receives();
prepost_untraversed_receives();
barrier();
begin_timer();
send_25_messages();
wait_for_25_responses();
end_timer();
clear_receives();
```

(a)

```
prepost_traversed_receives();
post_25_latency_receives();
prepost_untraversed_receives();
barrier();
wait_for_25_messages();
send_25_responses();
clear_receives();
```

(b)

Figure 4. Pseudo-code for pre-posted queue impact benchmark: (a) node 0, and (b) node 1

5.2. Simulation Environment

System-level simulation used a simulator based on Enkidu [21], a component-based discrete event simulation framework. This simulator was originally designed to model a homogeneous sea of PIMs as a supercomputer. To simulate a more traditional system with commodity processors for both the host and the NIC, `sim-outorder` from the SimpleScalar [8] tool suite was integrated into this framework. In addition, a network model was needed. For a simple two node benchmark, the network was modeled as a point-to-point connection with a fixed latency. For the NIC, components representing DMA engines were added. Properly modeling the interaction with the host processor also required a memory controller, DRAM chips, and a simple model of the interface between the host processor and NIC. To maximize fidelity, the memory hierarchy was modeled to include contention for open rows on the DRAM chips.

Both the NIC and the host processor used the PowerPC ISA, augmented with a basic subset of the Altivec [11] vec-

Parameter	CPU	Conv. NIC	PIM
Fetch Q	4	2	1
Issue Width	8	4	1
Commit Width	4	4	1
RUU Size	64	16	NA
Integer Units	4	2	1
Memory Ports	3	1	1
L1 (Size/Assoc.)	64K/2	32K/64	4K/8 (I)
L2	512K	none	none
Clock Speed	2GHz	500MHz	
Main Mem. Lat.	70-80 ns	110-120 ns	
ISA	PowerPC		PPC/Altivec
Net. Wire Lat.	200 ns		
Net. Wire BW	3 GB/s		

Table 1. Processor Simulation Parameters

tor instruction set. The semantics of this Altivec subset were changed to allow 256-bit vectors. Only six instructions were used (load vector, store vector, copy vector, compare equal-to, and vector bitwise AND) and only 8 vector registers were used.

5.3. Processor Models

The main processor was parameterized to be similar to a modern high-performance processor, such as an AMD Opteron. Although the Opteron is only six way issue, the SimpleScalar tool suite prefers powers of two and so an aggressive 8 way issue processor was modeled. The NIC processor was parameterized to be similar to a processor in higher-end network cards, such as the PowerPC 440 (see Table 1) that is used in Red Storm. A simple bus on the NIC connected the main processor with the DMA engine, SRAM, and matching structure. This bus was simulated with a 20 ns delay to access any components connected to it. Overall, this model attempts to provide as reasonable of a match to a real network as possible.

6. Results

Figure 5 compares the performance of a NIC using a conventional processor with a NIC using two PIMs for the message processing. The left side of the graph compares bandwidth (a product of message throughput) while the right side compares zero byte message latency. The PIM-based solution shows dramatic improvements in bandwidth when multiple messages are in flight. Even with a short posted receive queue, the overlap achieved by having two simple processors and multiple threads with fine-grained locking can be 10-20% on short messages. Moving to longer queues rapidly exposes the advantage of a wide SIMD unit and wide memory accesses and offers an order of magnitude better performance. As the length of the message grows, the time to DMA the message begins to hide the message processing time; thus, both approaches have the same asymptotic bandwidth. In general, the PIM-based approach ramps much more quickly.

The pure short message latency performance (the right side of Figure 5) is a much more mixed result. At short posted receive queues, the PIM loses dramatically. With short queues, the processing of a single message cannot be multithreaded effectively and cannot leverage wide memory accesses or wide ALU capabilities. Furthermore, there is no significant parallelism (between multiple arriving messages) to exploit the capabilities of PIMS. As the queue length grows, however, PIMs are able to better utilize their unique architectural capabilities.

The relatively poor latency performance of the PIM is explained by the simplicity of the PIM processor. With only a single message, the PIM is unable to spawn multiple threads to mask latency. Profiling indicates that during the latency tests the PIM spends 75% of its cycles with only the main thread active. The operations this thread performs cannot be parallelized due to the ordering semantics of MPI and to avoid deadlock. With only a single thread, the PIM is essentially a single pipeline without branch prediction or the ability to hide memory latency, competing against a dual-issue pipeline with a single-cycle cache. This disadvantage can negate much of the benefit of wide word comparison.

7. Conclusions and Future Work

Novel features that are core to many modern PIM architectures offer a dramatic advantage for the types of processing found in an MPI library. Exploiting these processors in the embedded environment found on high-performance network interfaces is a natural progression from current NIC architectures that use a traditional microprocessor. The benefits are numerous. The latency for a single message with long queues drops dramatically. More importantly, small message bandwidth grows by 10% with extremely short queues, $2\times$ with moderate queue lengths, and $20\times$ with extremely long queues. These advantages are conferred by the inherent advantages in PIM architectures.

Going forward, we plan to further explore how to leverage the capabilities of PIMs and how to enhance PIMs in the network interface. One avenue of exploration is more PIMs on the NIC. A second potential avenue of exploration is enhancement to the PIM core to make single threaded performance more competitive with PowerPC performance.

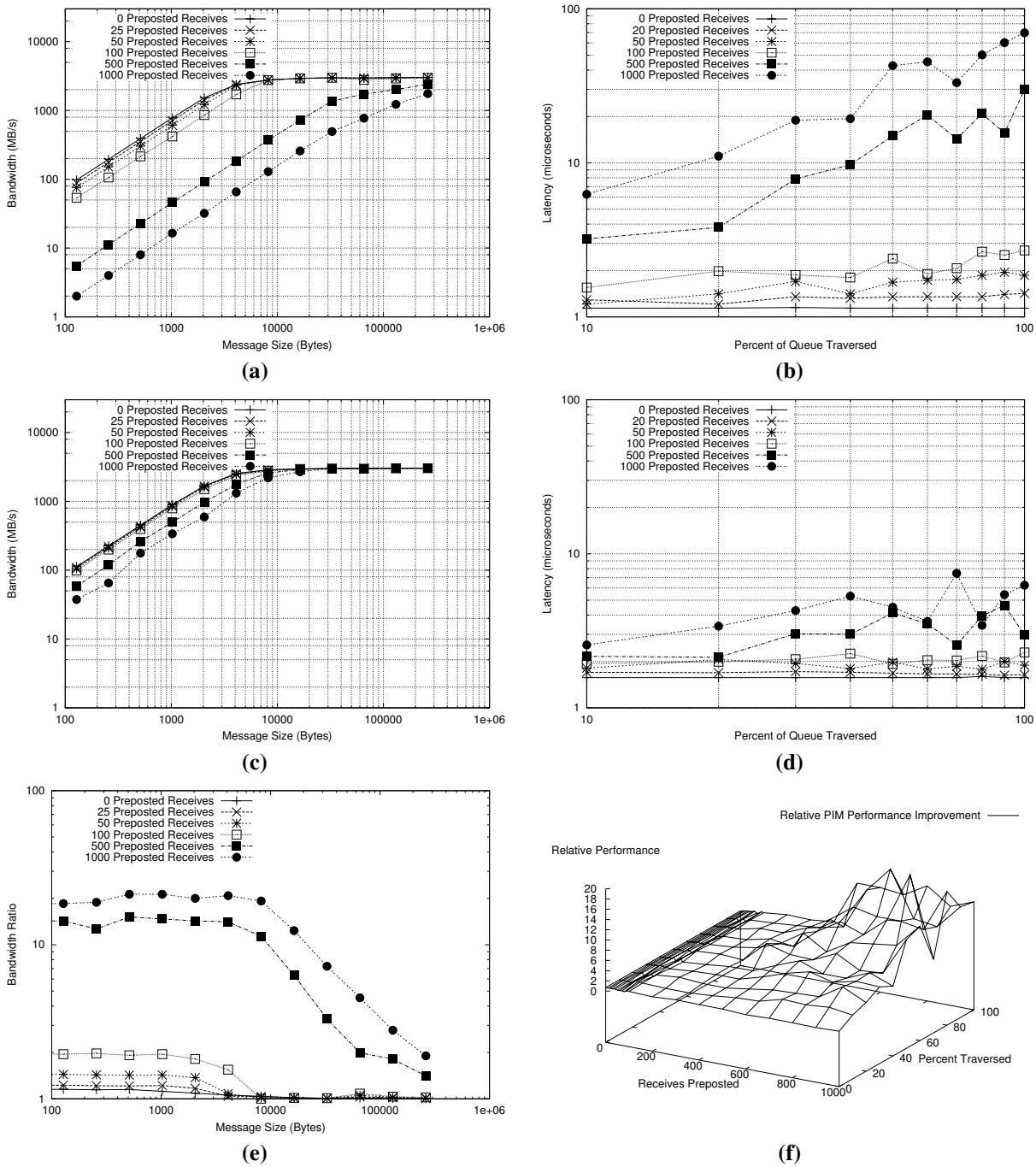

Figure 5. Conventional processor-based bandwidth and latency curves ((a) and (b)); PIM based-bandwidth and latency curves ((c) and (d)); Ratio between the two ((e) and (f))

References

[1] R. Alverson. Red Storm. In *Invited Talk, Hot Interconnects 10*, August 2003.

[2] N. J. Boden, D. Cohen, R. E. F. A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[3] R. Brightwell and K. D. Underwood. An analysis of NIC resource usage for offloading MPI. In *Proceedings of the 2004 Workshop on Communication Architecture for Clusters*, Santa Fe, NM, April 2004.

[4] J. Brockman, P. Kogge, S. Thoziyoor, and E. Kang. PIM lite: On the road towards relentless multi-threading in massively parallel systems. Technical Report TR-03-01, Computer Science and Engineering Department, University of Notre Dame, February 2003.

[5] J. B. Brockman, P. M. Kogge, V. Freeh, S. K. Kuntz, and T. Sterling. Microservers: A new memory semantics for massively parallel computing. In *ICS*, 1999.

[6] J. B. Brockman, S. Thoziyoor, S. K. Kuntz, and P. M. Kogge. A low cost multithreaded processing-in-memory system. In *3rd Workshop on Memory Performance Issues (WMPI)*, 2004.

[7] D. Buntinas, D. K. Panda, and P. Sadayappan. Fast NIC-based barrier over Myrinet/GM. In *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2001.

[8] D. Burger and T. Austin. *The SimpleScalar Tool Set, Version 2.0*. SimpleScalar LLC.

[9] W. J. Camp and J. L. Tomkins. Thor's hammer: The first version of the Red Storm MPP architecture. In *In Proceedings of the SC 2002 Conference on High Performance Networking and Computing*, Baltimore, MD, November 2002.

[10] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 1993.

[11] IBM Microelectronics Division. *PowerPC Microprocessor Family: Altivec Technology Programming Enviornments Manual*. IBM, 2.0 edition, July 2003.

[12] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas. FlexRAM: Toward an advanced intelligent memory system. In *Proceedings of 1999 IEEE International Conference on Computer Design*, Austin, Texas, USA, Oct. 1999.

[13] P. M. Kogge, S. C. Bass, J. B. Brockman, D. Z. Chen, and E. H. Sha. Pursuing a petaflop: Point designs for 100TF computers using PIM technologies. In *6th Symosium on Frontiers of Massively Parallel Computation*, Annapolis, MD, Oct. 1996.

[14] A. B. Maccabe, W. Zhu, J. Otto, and R. Riesen. Experience in offloading protocol processing to a programmable NIC. In *IEEE International Conference on Cluster Computing*, September 2002.

[15] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[16] Message Passing Interface Forum. MPI: A message-passing interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8, 1994.

[17] A. Moody, J. Fernandez, F. Petrini, and D. K. Panda. Scalable NIC-based reduction on large-scale clusters. In *Proceedings of the ACM/IEEE SC2003 Conference*, November 2003.

[18] R. C. Murphy and P. M. Kogge. Trading bandwidth for latency: Managing continuations through a carpet bag cache. In *Proceedings of the International Workshop on Innovative Architecture 2002 (IWIA02)*. IEEE Computer Society, January 10-11, 2002.

[19] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent DRAM: IRAM. *IEEE Micro*, April, 1997.

[20] F. Petrini, W. chun Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics network: High-performance clustering technology. *IEEE Micro*, 22(1):46–57, January/February 2002.

[21] A. Rodrigues. Enkidu discrete event simulation framework. Technical Report TR04-14, University of Notre Dame, 2004.

[22] A. Rodrigues, R. Murphy, P. Kogge, J. Brockman, R. Brightwell, and K. Underwood. Implications of a PIM architectural model for MPI. In *Proceedings the 2003 IEEE Conference on Cluster Computing*, December 2003.

[23] T. Sterling and H. Zima. Gilgamesh: A multithreaded processor-in-memory architecture for petaflops computing. In *In Proceedings of the SC 2002 Conference on High Performance Networking and Computing*, Baltimore, MD, November 2002.

[24] B. Tourancheau and R. Westrelin. Support for MPI at the network interface level. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 8th European PVM/MPI Users' Group Meeting*, Santorini (Thera) Island, Greece, Septermber 2001. Springer - Verlag.

[25] K. D. Underwood and R. Brightwell. The impact of MPI queue usage on message latency. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, Montreal, Canada, August 2004.

[26] K. D. Underwood, K. S. Hemmert, A. Rodrigues, R. Murphy, and R. Brightwell. A hardware acceleration unit for mpi queue processing. In *19th International Parallel and Distributed Processing Symposium (IPDPS'05)*, April 2005.

[27] J. Wu, P. Wyckoff, and D. K. Panda. High performance implementation of MPI datatype communication over InfiniBand. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, April 2004.

[28] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, March 1995.